5-2006

Nectar Clean 2006 A Complete Disk Utility Program

Matthew Areno
Utah State University

NECTAR CLEAN 2006
A COMPLETE DISK UTILITY PROGRAM

by

Matthew Areno

Thesis submitted in partial fulfillment
of the requirements for the degree

of

DEPARTMENT HONORS

in

Your Department

Electrical & Computer Engineering

Approved:

Thesis/Project Advisor	**Department Honors Advisor**
Paul Israelsen	Wynn R. Walker

Director of Honors Program

CHRISTIE FOX

UTAH STATE UNIVERSITY
Logan, UT

2006

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

ABSTRACT

The objective of this paper is to introduce a hard drive enclosure kit and program, named Disk Utility, which provides various utilities to a work-bench tech or a home user. Both the authors have worked in a tech-bench area and have found that this is a useful tool to have, as many times trouble-shooting type tasks are necessary to properly diagnose problems on a hard drive. It has also been found to be far easier to implement these changes on a hard drive that doesn't need to be moved from machine to machine.

Many design approaches are covered in this paper, with their various successes and failures. Although many design approaches were used, we will focus on the implementation that is now seen in our project.

In overview, we used Microsoft Visual Studio to develop and test the code. We used a .NET project to provide functionality for the GUI. Source code will be provided as necessary to illustrate ideas and objectives.

INTRODUCTION

As was introduced in the abstract, we have both worked in tech bench areas. Many times it becomes necessary to work on a faulty computer, and testing the software that has been installed on the machine is mandatory. It is also often the case that working from the faulty machine is undesirable and ineffective, hence the need arises to quickly remote access the hard disk

and test for difficulties. Many times we have seen that if a problem can be ruled out as a software glitch, then the requisite hardware can be replaced.

This is the reason for our project. Not only have we made it easy for the tech who would be working on a customer's hard drive to have a variety of tools at his or her disposal, but with the USB connection we have eliminated a needless step of installing the hard disk into another healthy machine in order to begin work. This remote access via a USB port is very handy, and can be taken from site to site if the tech that does the work travels.

In order to accomplish the goal of portability, we wanted the code to be able to be used in a Windows NT environment, with easy portability from machine to machine. From the beginning stages, we intended to use Visual Studio 6, but the necessary Windows NT implementation code was not present. This would make work difficult on a variety of computers, as Windows NT operating systems, such as 2000 and XP, are widely used.

The approach we settled on is a class system. Each GUI window has its own form with header functions as well as the necessary C++ implementation functions attached. Microsoft has provided many system variables and pointers that has made the goal of portability feasible, as well as streamlining our code.

The off-the-shelf unit used makes use of a simple USB to IDE controller devices and houses a hard drive in a metal case with a vent. This housing unit serves our needs perfectly, and draws power off the USB device.

As the report continues, the reason behind our decisions will be discussed and analyzed. We understand that the outcome of our decisions has both positive and negative effects, and we hope to communicate why the positives outweigh the negatives.

2. DESIGN GOALS

2.1 Problem Analysis

The driving force behind our project is the problem that many computer tech-support specialists need to access a drive of a computer remotely, that is, while the particular drive is not currently in the machine that might be faulty. As will be discussed further in the report in section four, we would like to make this an embedded system, to function as a sort of hard disk dishwasher, where several hard disks may be inserted into a multi-drive bay and have the appropriate diagnostics performed on them. For now, we work on one hard drive at a time.

Part of the problem is that many times a faulty computer might have malicious software installed. We recognize that if the hard disk can be isolated from the machine that houses it, it will be easier to stop malicious software. This is what drove us to have an external enclosure unit rather than just develop the code.

2.2 Constraints

The biggest constraint that we had when designing our project was making sure that our code had something else to offer other than what was already available. We recognized that although there are many tools available, it is difficult to get them all in one place to be used. We needed to make sure that our project provided an environment where the tech could have commonly used tools in one place, with point and click action to implement changes.

Another constraint we had was that the final product needed to be portable. This constraint has inherent advantages, as have been discussed. Our code fits on a jump drive, and the enclosure kit allows us to be able to move about unfettered.

3 DETAILED DESIGN

The design of this tool is broken up into six main function, each performing different diagnostic or analytical functions. An additional idea was to create a way for the user to define other specific functions that may be invoked through a drop-down menu rather than buttons, but we feel that the tools we have included provide sufficient capabilities to work on hard drives. These functions are listed below.

1. Copy
2. Format
3. Defrag
4. Clean
5. Virus Scan
6. Image

Each of these six tools will now be discussed in detail, specifically how each was implemented.

3.1 Design Overview

The main window that is seen when NectarClean 2006 is executed is shown below in figure 3.1. We leave a little extra room to the side of the directory window for pie charts and other images that aid the user in understanding disk contents. Pie charts and other such various graphs can be displayed here. The functionality code behind this part of NectarClean is the hub of all the commands. Each button and each command from the menu bar contains

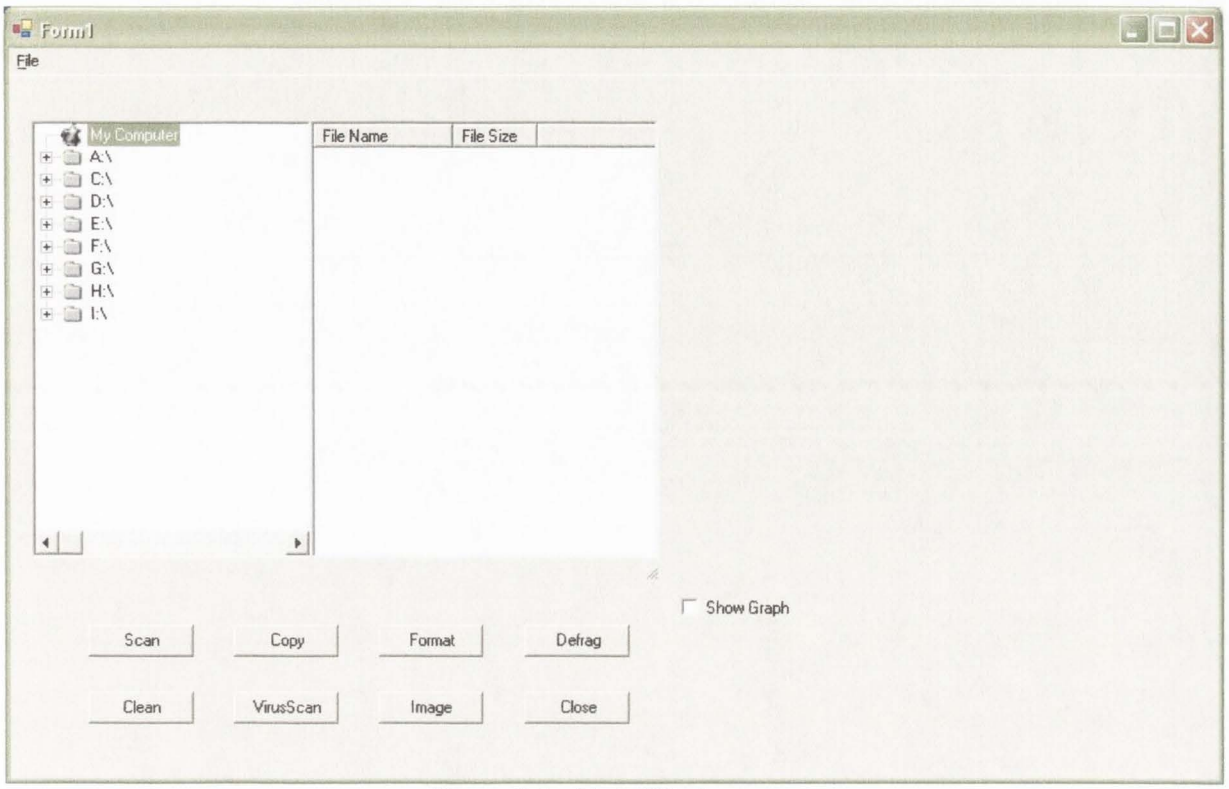

Figure 3.1. Main Window

a private member function in the file Form1.h, which is the default name for the default window of a GUI program in Microsoft Visual Studio .NET 2003. For testing purposes, we made extensive use of try and catch blocks in order to make sure that there were no user contributed errors involved with the use of the program. From this window, the user can select and commence various diagnostic or other works on a given drive. One reason we decided to have NectarClean run natively on the computer rather than as an embedded system on the enclosure unit is because of the vast resources a home computer offers for multitasking. From here we will investigate the various tools offered by NectarClean.

3.2 Scan Function

The scan function is invoked automatically whenever the program starts. Its primary purpose is to simply scan the computer and report the logical drives found on the computer. This is done by making use of built in Windows namespaces, such as System::IO and System::Diagnostics. Within these namespaces are the classes Directory, DirectoryInfo, File, and FileInfo. These 4 classes are accepted to find directories and files that are contained within a specific path. Therefore, when the program begins, the logical drives are found and displayed to the user. This gives the user a foundation to begin execution of tools on the drives currently available. At any time, the user may hit the scan button to send the directory listing back to the logical drives.

3.3 Copy Function

Using the directory structure presented in the Scan function, the user may first select a source directory, or in other words, a directory to copy. The currently selected directory can be seen in the status bar directly below the directory tree. When a user selects the copy button, a new directory window shows up and gives the user the option of selecting the destination directory.

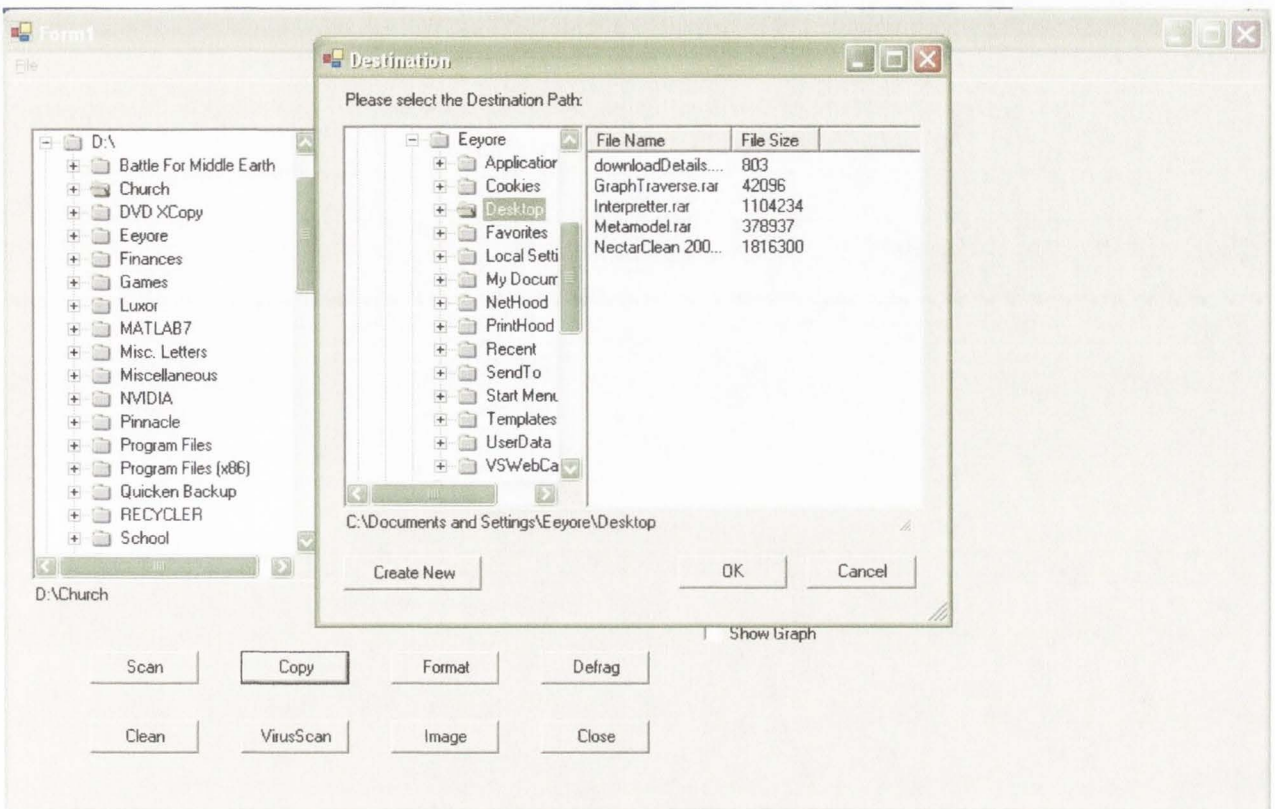

Figure 3.2. Copy function.

When the selected files or folders are completed, they can then be copied to the destination folder. This is shown in figure 3.2.

To allow this functionality, the TreeView, which presents the directories within the current path, is constantly updated. This is done anytime an object is selected, expanded, or contracted. Once the folder to be copied has been selected, the user may push the Copy button. If no node is selected, the user will be presented with an error.

Once the Copy button has been push, the current directory is recorded and the Destination dialog is started. This presents the same functionality

and implementation as the main function, which allows the user to select any destination folder they wish. There are similar safe-guards as those previously listed, but also include safe-guards against using a file as the destination directory.

Once the source and destination directories have both been established, the Transfer dialog is instantiated. Within the Transfer operation is the actual code to perform the copy. The first step is to create the appropriate folder in the destination path, if it is does not already exist. Once the folder is created, or found to exist, a list of all the files contained within that directory is performed. With a list of all the files within the directory, the program can begin copying the files. This is done through the File::Copy function within the System::IO namespace. Each file is first checked to see if it exists. If the file does exist, a window pops up asking if the user would like to overwrite the file. The user is present with the options "Yes, Yes to All, No, No to All". According to the user's selection, file copying continues. Once all the files within the directory have been copied, a list is created of all directories contained within the original directory. The function is then recursively called with each of the directories listed. This ensures that the function will copy all files and folders within the directory, regardless of hierarchy levels.

In order to make the user aware of the status of the transfer, a progress bar is used to track the status. Prior to each file be transferred, the size is determined. Once the file is copied, its size is collected and added to the total amount transferred. This is then compared to a calculation performed previously that received the size of the directory. Every time the percentage transferred increase by a minimum of one percent, the progress bar is updated. This allows the user to know what the current status is.

Once the transfer has completed, control is returned back to the main function which may then return to normal operation.

3.4 Format Function

The first step in this process is gather the drive and path from what was selected in the main window on NectarClean. Again, we make an attempt to catch user errors by making sure that a drive is actually selected, as well as confirming the selection with a Confirmation window. This is a separate file called Confirmation.h that halts the process that is currently running and makes sure that the user is sure that he or she wishes to

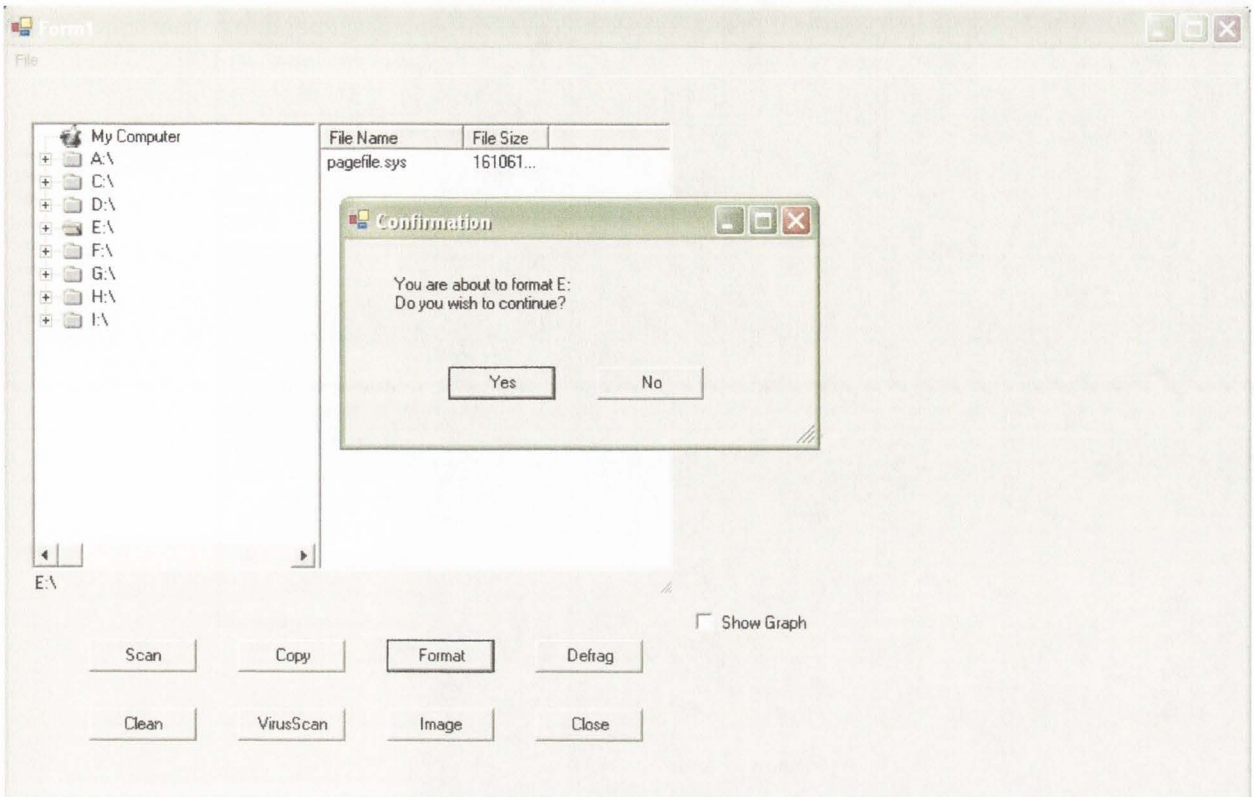

Figure 3.3. Format Function

proceed. The target drive to be formatted is shown, and the user is left with the choice to continue. Then the native Windows format.com is called, and the drive is formatted. The dialog for this is shown above in figure 3.3.

The Windows environment allows us to spawn a process by using the Process::Start command. This command is used heavily for all parts of our program that need to interface to outside programs and applications.

3.5 Defrag Function

Defrag performs a defragmentation of the drive selected. Although Windows does not inherently support automatically starting a

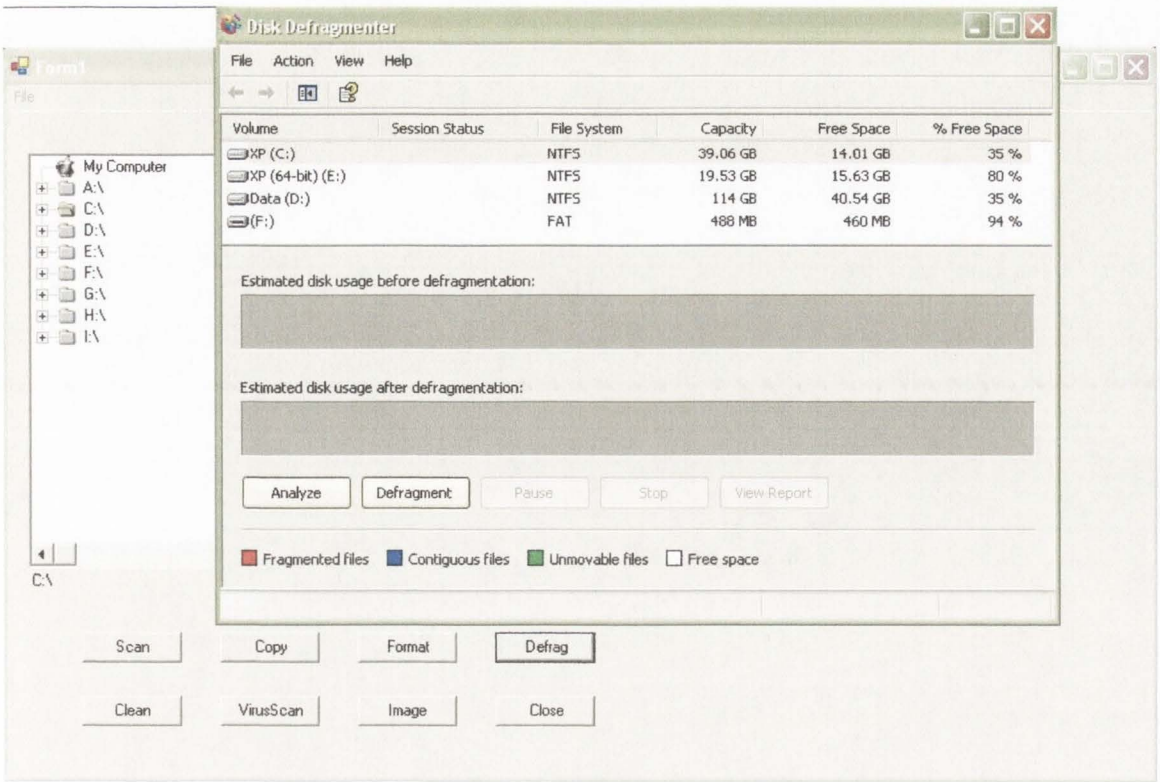

Figure 3.4 Defrag Function

defragmentation any way other than via the command prompt, we were able to find a way to invoke the program and send it the drive letter, which results in what is shown above. The defragmenting does not start automatically, but the drive requested is automatically selected and the user simply has to push the Analyze or Defragment button. This is shown above in figure

3.6 Clean Function

The clean function is the most intensive function of the group. The idea behind the clean function is to, obviously, clean the selected drive. In

order to most efficiently clean a computer, we determined the following items should be addressed.

1. Startup folders
2. Web browser caches
3. Web browser cookies
4. Web browser history
5. User temporary data
6. Windows temporary data

Although cleaning these directories and files does a sufficient job in cleaning the computer, additional functions could still be performed. The main functionality that we intend to address in later work is cleaning the registry, specifically the Run and RunOnce folders within the registry under the Windows\CurrentVersion entries. Performing this task is relatively simple on a drive that is currently booted into Windows, but for a drive that is not currently active this involves the parsing of ntuser.dat files, which is beyond the current project. The checkbox menu presented below in figure 3.5 is shown to the user right after they click on the clean button.

3.6.1 Clean Startup Folders

The first option given to the user is to clear the startup folders. In order to do this, a directory listing is obtained for <selected drive>\Documents and Settings. This directory listing provides the function with a listing of all the

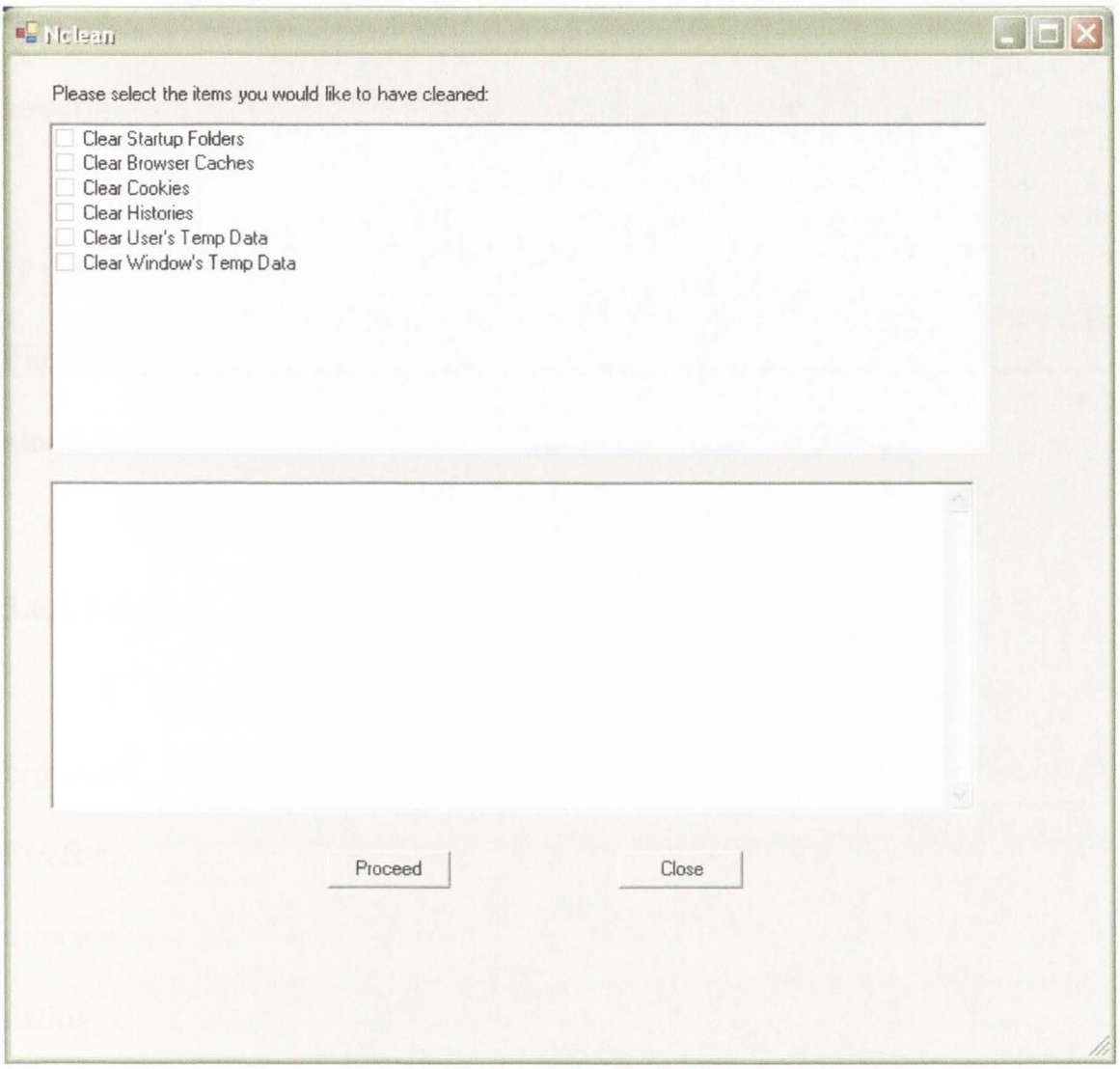

Figure 3.5 Clean Function

users on the drive. Once the users are determined, the startup folders may be scan by accesing:

<selected drive>\Documents and Settings\<user>\Start Menu\Programs\Startup

After accessing this directory, a file listing may be obtained to determine what is executed when the Operating System is booted. A list of these items is then presented to the user after clearing the current list. The user may

then select the items they wish to have removed. Once they have made their selections, they may hit the proceed button again to begin cleaning the drive.

All of the selected items are first accessed to see if they are checked. If they are, the selected items are removed from the directory using the File::Delete function within System::IO. Any items not selected are left alone.

3.6.2 Clean Web Browser Caches

This program currently supports cleaning of three different web browsers, namely Internet Explorer, Netscape Navigator, and Mozilla Firefox. Each of these three browsers stores their information is separate directories. Similar to what was done in cleaning the startup folders, a listing of all users is retrieved by scanning the Documents and Settings directory. Once a listing of all users is obtained, they are all scanned looking for the appropriate caches. These are located in the following directories.

Internet Explorer
<drive>\Documents and Settings\<user>\Local Settings\Temporary Internet Files

Netscape Navigator
<drive>\Documents and Settings\<user>\Application Data\Netscape\NSB\Profiles\Cache

Mozilla Firefox
<drive>\Documents and Settings\<user>\Application Data\Mozilla\Firefox\Profiles\Cache

After retrieving the folder information, a list of all files and directories is obtained, and similar to the copy functions, it recursively calls itself only this time to delete the files instead of copying the files. Some files are protected by the Operating System, or may currently be in use. Our program handles this by catching any exceptions that are thrown and bypassing the deletion of the current file. Once all the users caches are empty, the program continues.

3.6.3 Clear Web Browser Cookies

The functionality behind this portion is identical to clearing the cache, only with using the following directories:

Internet Explorer
<drive>\Documents and Settings\<user>\Cookies
Netscape Navigator
<drive>\Documents and Settings\<user>\Application Data\Netscape\NSB\Profiles\<profile>
Mozilla Firefox
<drive>\Documents and Settings\<user>\Application Data\Mozilla\Firefox\Profiles\<profile>

Within the Netscape and Mozilla directories is a file titled cookies.txt. In order to remove the cookies.txt file, which must be maintained in order for the browser to function properly, the file is deleted and a new file is created. This file is a cleaned version that is compatible with both web browsers. This allows the cookies to be reset without affecting the functionality of the web browsers.

3.6.4 Clear Web Browser History

As before, the functionality is again similar to that of the cache clearing. The directory locations for the histories is shown below.

Internet Explorer
<drive>\Documents and Settings\<user>\Local Settings\History

Netscape Navigator
<drive>\Documents and Settings\<user>\Application Data\Netscape\NSB\Profiles\<profile>

Mozilla Firefox
<drive>\Documents and Settings\<user>\Application Data\Mozilla\Firefox\Profiles\<profile>

Again, as before, the Netscape and Mozilla browsers have a file located within the profiles directory called history.dat. This file, however, is not critical to the functionality of the web browser, so it may simply be removed. The Internet Explorer history is a collection of folder instead and must be appropriately traversed in order to remove all data.

3.6.5 Clear User Temporary Data

This function is used to remove all temporary data contained within the User's profile. This data may be accessed through the following directory:

<drive>\Documents and Settings\<user>\Local Settings\Temp

The elimination of this information again requires a traversal of the directory to remove all files and folders contained therein. Once every directory is

emptied of the files it contains, the folder is deleted. If files still exist within the folder, it is instead left alone.

3.6.6 Clear Windows Temporary Data

This function is nearly identical to the Clear User Temporary Data function, only using the following directory.

<drive>\Windows\Temp

As before, the directory is traversed and all files and folder deleted.

3.6.7 Clean Completed

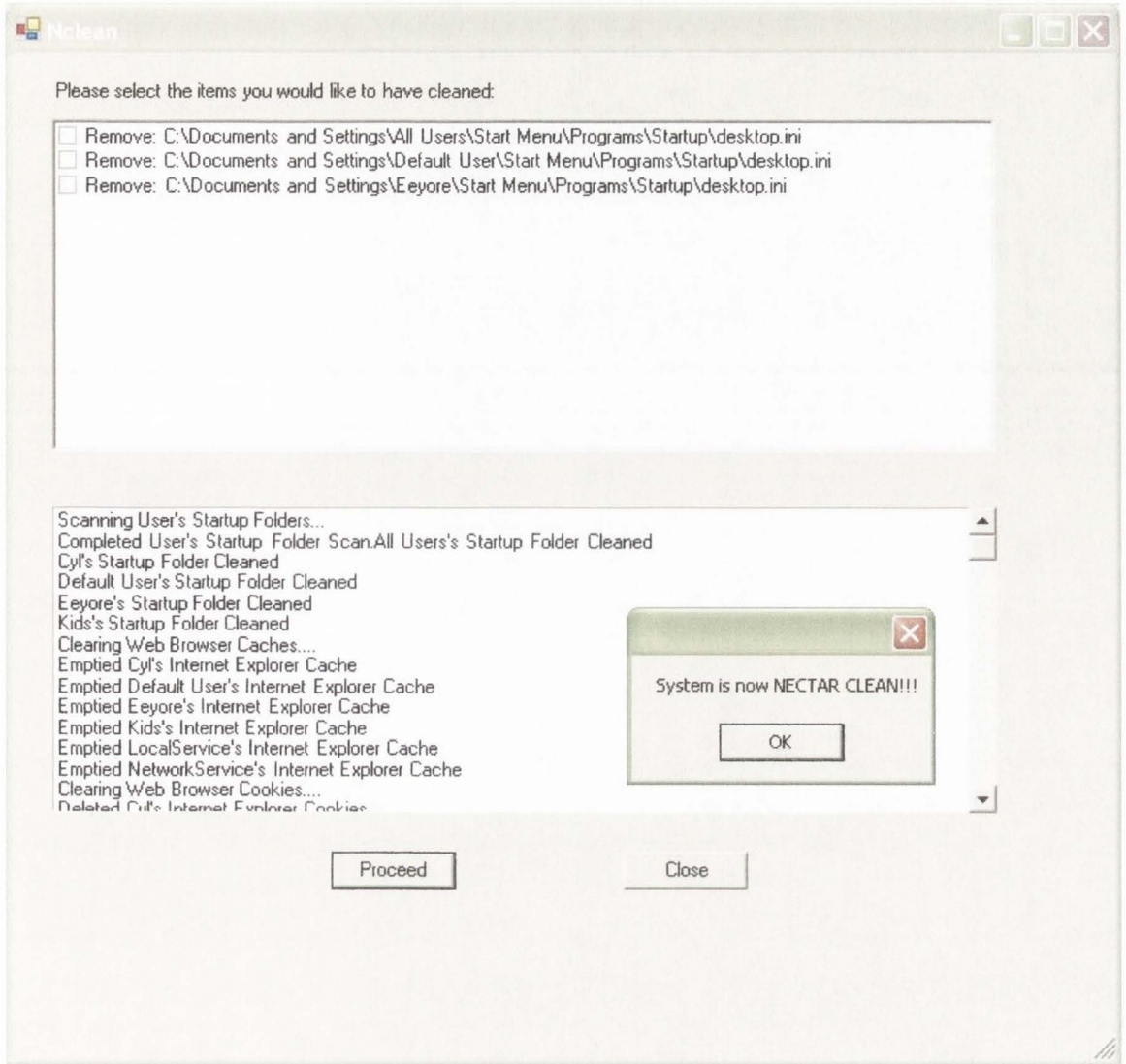

Figure 3.6. Clean Completed

Figure 3.6 above shows the output of the program after cleaning a hard drive. The second box shows the result of each of the functions previously discussed. In addition, a box is popped up that displays when the cleaning has completed, or if a major problem occurs, shows the resulting error.

3.7 Virus Scan Function

The purpose of the virus scan is to do exactly what it states, scan a drive or directory for viruses. Figure 3.7 is the output of the implementation of this function.

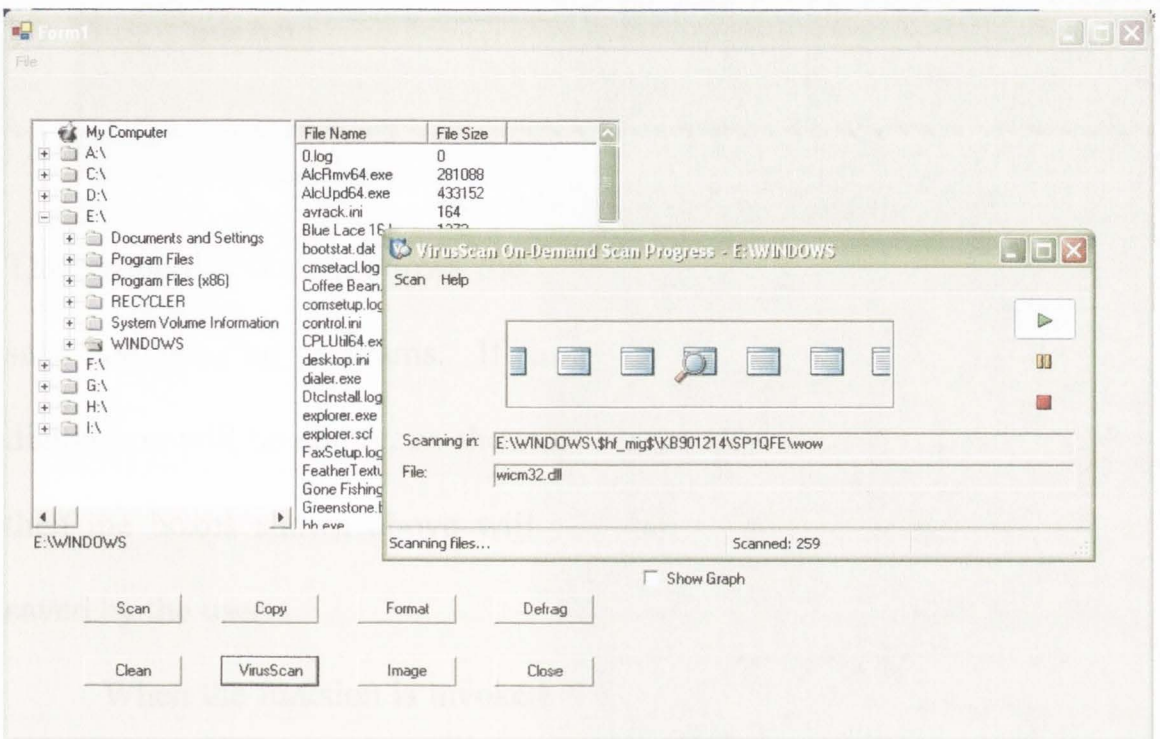

Figure 3.7. Virus Scan Function

The virus scan function is implemented by having the user define the location of their virus scan program. This is done through using the properties function. This is shown below in figure 3.8.

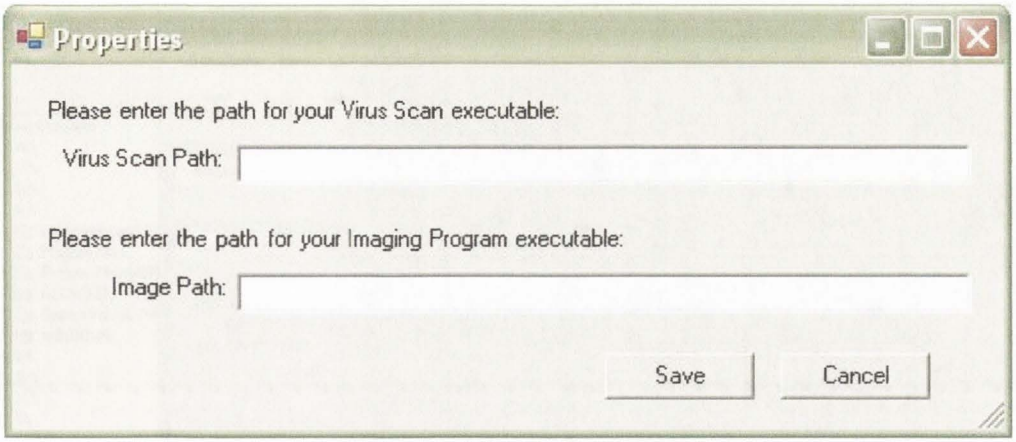

Figure 3.8 Properties Menu

This properties dialog, allows the user to input the location for their virus scan and imaging programs. If the properties have not been set yet, the dialog box will be empty, as shown above. If the properties have been set, then the boxes shown above will be populated with the previous values saved by the user.

When the function is invoked, the values set in the properties are read and the corresponding file is called. In addition, the selected drive or folder is passed as a parameter to the program. This results in what is shown above in figure 3.7. The selected directory was E:\Windows and when the virus scan program was started, this directory is what was scanned.

3.8 Image Function

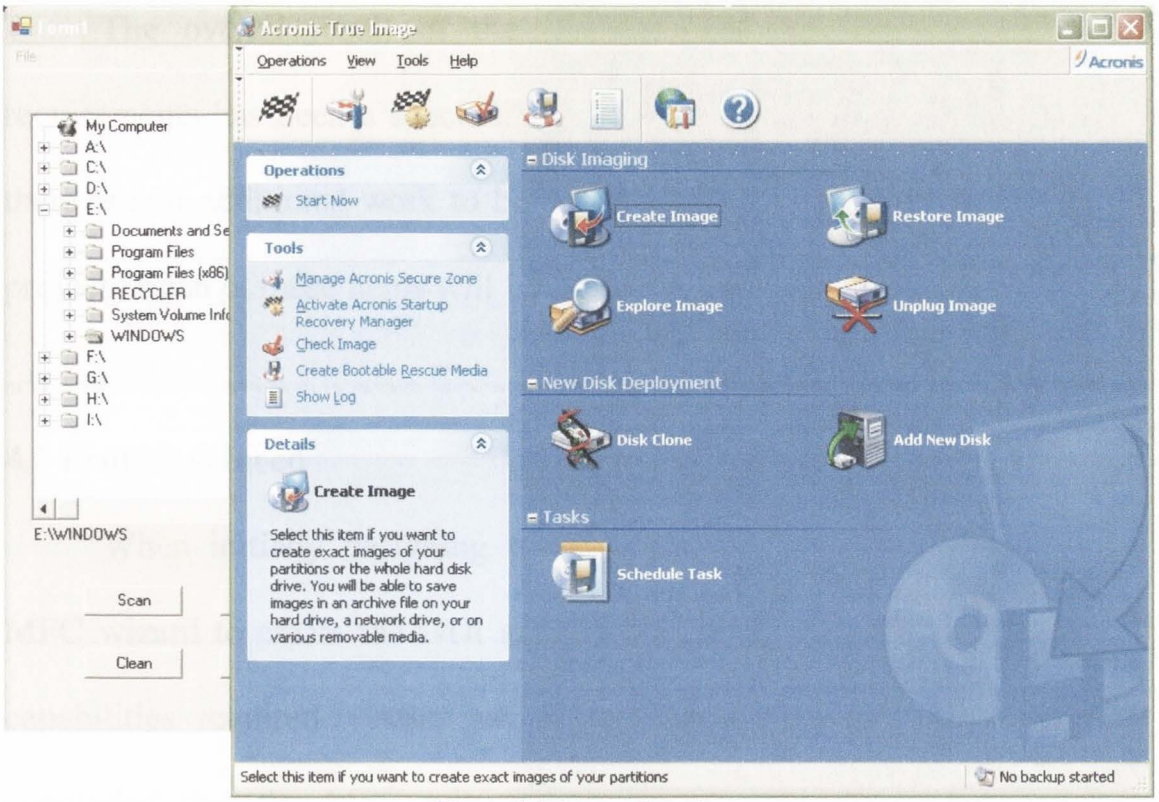

Figure 3.9 Image Function

The implementation of the image function is almost identical to the virus scan function. It simply reads from the properties specified by the user to determine the imaging program and then implements it.

The purpose of an imaging program is to make a backup copy of a drive or directory. The imaging program shown above is Acronis True Image which we determined to be the best available. It allows the user make a backup copy of their drive even if that drive is currently active. This produces a large advantage over the copy function which has difficulties copying files that are currently active.

4. RESULTS

The overall result of the program in comparison to the initial requirements has been a success. Although there have been problems and there is still additional work to be done, this has been a largely productive project. Each of these items will be discussed in more detail in this section.

4.1 Problems faced

When initially designing the program, we used Visual Studio 6.0 MFC wizard to create the GUI and the functions necessary to provide the capabilities required. After several days of frustrating work, we soon concluded that the MFC wizard was not able to provide us with the functionality that we required for this project. We therefore decided to switch to Visual Studio .NET 2003 which has the built in Windows Forms library to create the GUI and the System namespace which has a very large variety of file and directory functions. This change greatly simplified the design process actually allowed us to easily do a few things that we had not planned on implementing, such as automatic thread usage through system calls.

The second problem we faced was with an extra item we planned to add to the cleaning process. In our initial creation of Nectar Clean, we were able to perform a registry scan to look for items that are automatically

started when the computer first boots up. Within the Current_User and Local_Machine registry entries is a folder called Run and RunOnce, which are within the Mircrosoft->Windows->CurrentVersion root under the previously listed registry entries. This is a popular area for spyware producers to place a link to their programs so that they begin collecting data whenever the computer starts. Exploring these items and allowing the user to alter them is simple on a hard-drive that is currently active, meaning not just powered up, but actually running Windows. On non-active drives, this becomes a difficult problem.

Within Windows 2000 and XP is a file called NTUser.dat. This file exists for all users with a profile on the computer. The computer parses this file and reconstructs the registry for the current user whenever they log on. Therefore, in order to alter the contents of this file when it is non-active requires the ability to correctly parse this file for the needed information. Given this problem, we were unable to add this in the current version of Nectar Clean. We have successfully implemented a version that works on the active drive of a computer and put it into a console application, but due to the fact that it only works on the active drive, we felt it would not be appropriate to add it to the full version of Nectar Clean until we could

correctly parse the previously listed file to allow it to work for all drives, regardless of current status.

4.2 Additional Work

Currently the pie chart represents the size of each directory in the specified path, however it contains no labels showing with directory is represented by which color. It is possible to determine that if you realize where the pie chart starts, but to make it more user friendly, we plan to implement a color scheme so that we change the color of the directories listed to reflect there color in the pie chart.

Another item that we still plan to work on is the threading of the application. Some functions return immediately after invoking them, such as virus scan, image, format, and defrag. However, the copy function and the pie chart function require a large amount of time to complete and causes the program to halt until these operations are completed. For this reason, we plan to also change these operations to be conducted as a thread of the main program that will not prohibit the main function from continuing to execute.

The final item that we plan to include is the ability for the user to add custom diagnostic programs into Nectar Clean. We plan to use a drop down

menu that contains all the custom tools they have currently included in order to accomplish this. Along with the drop down menu bar will be add, remove, and execute buttons that will invoke a dialog box, similar to the properties dialog box, which will allow them to give the tool a name and a path to its executable, as well as actually invoking the program.

5.0 Final Scope of Work Statement

This project has been quite a challenge for the both of us. While we have both exhibited an interest in programming, it has always been had its end in electrical engineering applications. We were somewhat remorseful that we didn't have the time and resources to make this an embedded system that would make multi-tasking much easier, although we are content and satisfied that we have met the requirements for this project.

It has been beneficial to learn a new programming environment, which adds to our array of knowledge and understanding of driving the hardware that we will eventually design in our respective places of employment. We also have benefited from learning the Windows environment commands, which will apply to any Windows XP GUI program that we could participate in developing in the future. When we first sat down to do the project using Visual Studio 6.0 and used MFC AppWizard,

we had quite a bit of trouble running processes, and accessing components of the Windows operating system because the language wasn't streamlined to interface that well with it. When we made the switch from Visual Studio 6.0 to Visual Studio .NET, we found that we had a much easier time implementing system commands from the code. As engineers, we realize that we will never fully separate hardware design from software design, and so we carry a healthy respect for what our products might have to interface with, and what will be needed to drive the hardware. This is probably the most beneficial aspect of our project; we understand how to run our hardware.

When we began this project it was our hope that we could make this an embedded system. We quickly saw that the scope of such an undertaking was far beyond what would be considered as requisite for a senior project. We would have liked to have had more time to tinker with the project and implement a completely separate microcontroller with necessary memory and interrupt service routines. We have both had an in depth experience with embedded systems, and since Matt has done research in that field, this would have posed a very formidable, yet very interesting task to complete.

As we completed our embedded systems class halfway through our senior project, it would have been difficult to know how to accomplish our

goals. If we had to do it all over again, we would have chosen a more down to earth goal and stretched ourselves in areas with which we felt a little bit more comfortable. However, as has been mentioned, we are happy with the end product and our only lament is that we didn't have more time to give to the project.

5.1 PROJECT MANAGEMENT SUMMARY

This section will heavily refer to the Gantt chart shown in figures 5.1 and 5.2. We divided up the Gantt chart for this report into Fall and Spring We will report on how the work structure broke down and how tasks were delegated between the two of us. Delegating tasks and scoping out the work ahead of time was a beneficial way to prepare ourselves for the real world, more especially project management. We were able to see just how dynamic the work schedule can be. Several times we were all set to complete a task, and a distraction of some nature would steer us off course. Other times the different outcome of one task would completely change the outlook of all subtasks underneath it. What we will present here is the final breakdown of how things were accomplished. We're not sure we could present all deviations from the initial plan, and we have also included in this report our reasoning behind all our decisions anyhow.

Task management was decided lightly before the scope of the entire task was determined, and help was solicited when it was necessary. Many times Matt took over most of the code, as he has a more firm grasp on C++ than Chris does. For code research related tasks, Chris took over more often than not as he had more need of understanding than Matt. This prepared us both to understand the scope of our project fully.

Now we break down each task shown in the Gantt chart and offer a more full explanation of what the task entailed and the purpose why Matt or Chris was assigned it. The Gantt chart itself is broken by month and then by week.

Task:

1.0 Component research – This task was first on our list after we had selected the idea of the hard drive enclosure unit and the program to drive it, NectarClean. We researched Windows programming, evaluating our previous knowledge of it, and decided to make sure that the program was an intuitive GUI. We made sure that our program would be portable by finding the enclosure unit that we did.

2.0 Windows Programming – Because we had been through our programming classes together, we felt that we had reason to be confident

in our Windows programming from MS Visual Studio 6.0. As we've said, we were incorrect in this assumption and needed to do more research in January when we decided to develop NectarClean in MS .NET, which has a different interface to Windows GUI's.

3.0 Windows OS Research – When we were comfortable with programming in .NET, we needed to make sure we understood how to make system calls and call other GUI programs in order to run NectarClean.

3.1 System Calls – For the commands in NectarClean such as Copy, or even a subroutine task like TreeView, we needed to understand how Windows made system calls. Our information came from books we found at a local bookstore.

3.2 Calling other GUI commands – Calling system commands such as Copy or TreeView was a bit different than calling the Defrag program that was native to Windows. This area of research involved understanding how to use these programs within the control of NectarClean.

4.0 Budget – When we set out to do this project, we had a tentative list of things that we would need to purchase, including the research materials and what we thought would go into the final product. We both worked on this, simply because we both shouldered the financial load.

5.0Time Management – We had two big intervals of time management, updating along the way. During the fall, we would meet weekly and discuss our options and choose from them. When the spring we met again and updated our goals of when we wanted to complete certain tasks and have them ready. We didn't always meet our goal, but nothing came as a surprise to us because of meeting so often.

6.0Task Management – This was an ongoing process. When new problems or solutions arose, we were reasonable with each other and shared the tasks equally.

7.0Coding – The bulk of our project is coding, and so when we had decided on what functions we wanted to put into our program, based on our research, we began work. Many times Matt began the work and Chris finished it up, so that we would both have a hand in the coding.

7.1 Copy – This function of NectarClean was one of the first that we wanted to add, and so we began with it. It involved many of the ideas, such as function calls and process threads, which come up in the rest of NectarClean. It was a good starting point.

7.2Scan – This was an equally good starting point, because the very first thing NectarClean does when it starts up is fill the directory view of folders and files. We both worked on this part.

7.3 Image – Next came the imaging part of our code. We won't go into the explanation here, but we again worked jointly on it.

7.4 Format – Same as Image

7.5 Properties – Matt did most of the work on this part. We had to look into setting variables that could be stored in such a way that they would keep their values after the program exited. When we had figured this out the rest of the task was simple.

7.6 Clean – This was a very specific and formidable task. Our program takes its name from this project. Matt had already done some work on a command-line interface to NectarClean that he had used while working at Utah State's Help Desk. Much of the work for this task came from changing the user interface from command-line style to graphical style. Work was tedious, and involved scrolling through a lot of old code to see where it communicated with the user. We made sure that the command-line style was updated to fit with our new version of NectarClean.

7.7 VirusScan and Defrag – This task involved understanding how one program can call another within the scope of the former. It was easy to call Defrag as a system command in a command-line interface, but this isn't practical nor up to today's standards of

programming and expectations. We searched our resources, such as the internet and our programming books and saw a way around calling Defrag as a command-line style, and instead called it as it is now, a GUI. When this task was complete it was simply a repetition of it to use VirusScan within the control of NectarClean.

7.8 Putting it together – This part of the program was done all along the way, but really materialized at the end. As we built the GUI, we built the objects that the main window of NectarClean calls, and so when we were done, we added code to make it function smoothly together.

8.0 Testing and Updates – For a programmer, this task doesn't merit attention. It's part and parcel to coding. Anytime we had an error or didn't see the correct results of what we thought our code should be doing, we took a moment, tested it, debugged it, and updated it.

9.0 Writeups – This part is split up into the preliminary and final design reviews. Each person had a part in writing them.

9.1 When the time came for both the preliminary design review, Matt did a lot of the write-up and preparation of the PowerPoint slideshow for Dr. Israelsen.

9.2 Chris did a lot of the final design review PowerPoint presentation and we both contributed equally to this final report. Who wrote what in this report was largely determined to spear-headed work on that particular task.

10.0 Poster Design – Chris took care of the poster design, because of experience in web-design and desktop publishing.

11.0 Final Report Presentation – We shared this load equally, making sure to illustrate that we both knew what the other had done for the entire project.

Now that all the tasks have been explained, below you find the Gantt chart.

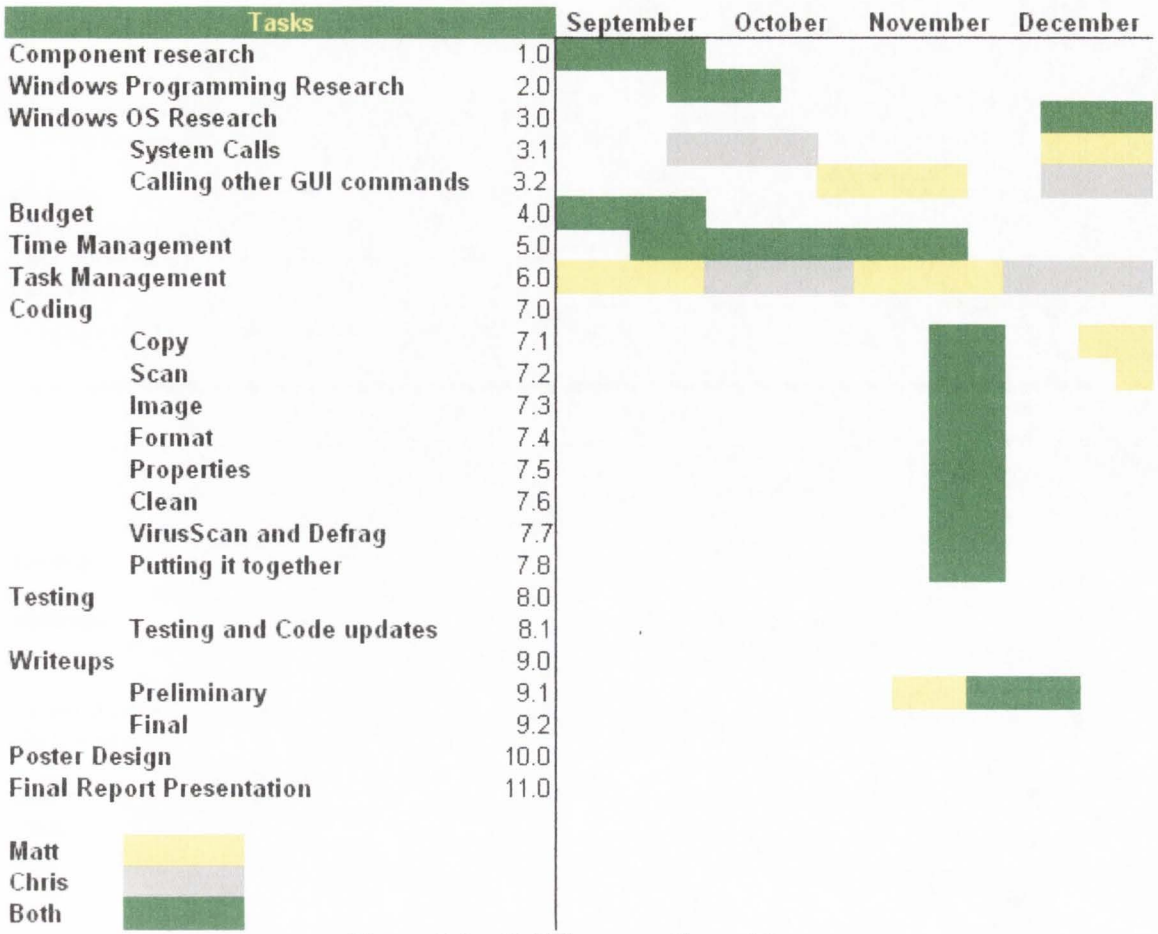

Figure 5.1 – Fall Semester Gantt Chart

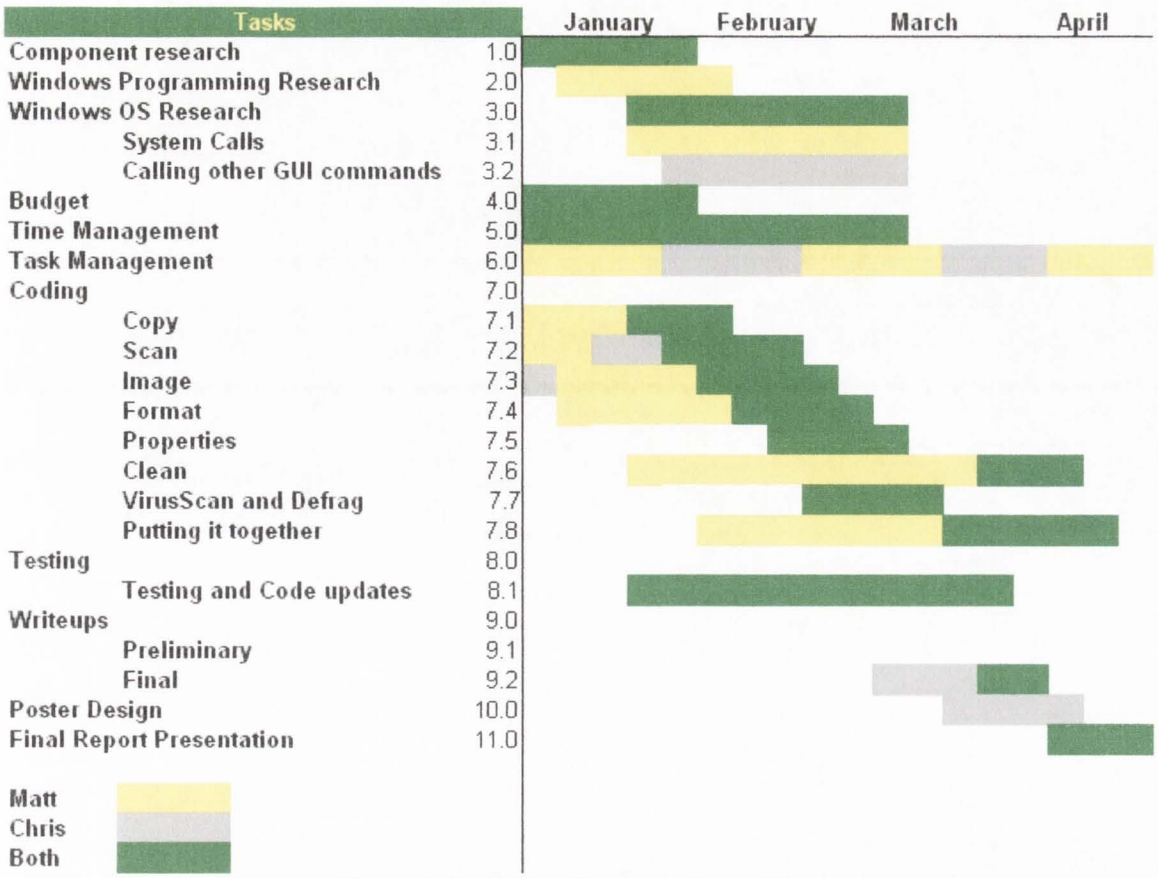

Figure 5.2 – Spring Semester Gantt Chart

BUDGET REPORTING

Another work-environment experience is reporting on the budget. We used the same file for our budget throughout the project so that we could show ourselves our target pricing and then the final result, in order to show what our time was worth. On items with a fixed price we were very successful, where we fell short was not necessarily in budgeting with money, but with time. We had a difficult time pinning down just how much time we spent, and so we were over budget with that respect. This goes to show just how much wisdom is necessary in time management. For the monetary

figures, we valued our time as a research assistant might be paid here in Logan at $10.00/hour. We count testing and debugging as final steps, not the normal testing and debugging that we normally do when we program. We ended up being slightly over budget because of our time, not because of the cost of necessary hardware. We drew the conclusion that this is why engineers are salaried.

Here is our budget, with target and actual costs:

Budget		Target		Actual	Itemized Total
Research					
Books		$40		$38.99	$1.01
Windows GUI Progamming	48 hours	$480	50 hours	$500	$20
System Calls and Programs	48 hours	$480	45 hours	$450	$30
Coding					
Functionality	450 hours	$4,500	500 hours	$5,000	$500
Final testing	15 hours	$150	20 hours	$200	$50
Converting Nclean	150 hours	$1,500	135 hours	$1,350	$150
Debugging	15 hours	$150	20 hours	$200	$50
Write ups	24 hours	$240	24 hours	$240	$0
Presentation	24 hours	$240	24 hours	$240	$0
Power Subsystem		$250		$0	$250
Enclosure Unit					
USB Cable		$25		$35	$10
Hard Drive Enclosure		$50		$40	$10
Hard Drive		$0		$0	$0
	Total	$8,105		$8,293.99	
		Over/Under:		$188.99	OVER

Figure 5.3 -- Budget

6. CONCLUSION

After completing this design, we have found that it has a lot of potential to become a great tool for hard drive diagnostics, but still requires a large amount of work to make it optimal. It does still have the potential we initially saw in it, meaning that we do still believe that it could be

implemented onto a custom drive enclosure. Certain applications, such as those specified by the user would still require the use of the main computer, but the majority of the programs contained within Nectar Clean could easily be implemented on an on-board microcontroller. The GUI interface created for Nectar Clean would still be used to allow the user to invoke certain tools on a drive, but the operations themselves would be carried out by the microcontroller in the enclosure.

In conclusion, we therefore feel that the project has been a great success in the overall effort to create a stand-alone disk utility enclosure system. Additional research and funding will be required in order to finish creation of this device, but the design is definitely possible.